Università degli Studi di Parma
Dipartimento di Ingegneria e Architettura
Sistemi operativi e in tempo reale – a.a. 2022/23

Semafori e mutua esclusione

Mutua esclusione mediante *lock* e *unlock*

- Mutua esclusione tra sezioni critiche della stessa classe:

Ti	Tj
…	…
lock(x)	lock(x)
<sez. critica A>	<sez. critica B>
unlock(x)	unlock(x)
…	…

Uso di lock e unlock

- Il meccanismo di lock-unlock soddisfa i principali *requisiti di correttezza* della mutua esclusione:
  - Garanzia di mutua esclusione, indipendenza delle sezioni critiche, assenza di deadlock
- Soddisfa inoltre i *requisiti di tipo sistemistico*:
  - Trasparenza tra processi/thread e indipendenza dal loro numero
- Non soddisfa i *requisiti legati all'efficienza*, ed è pertanto limitato al caso di sezioni critiche brevi:
  - Presenza di attese attive, interruzioni disabilitate
- *Non impedisce la starvation*, anche se non la determina
- *Non impedisce comportamenti errati* dei thread all'interno delle sezioni critiche brevi

Per un meccanismo di mutua esclusione generale

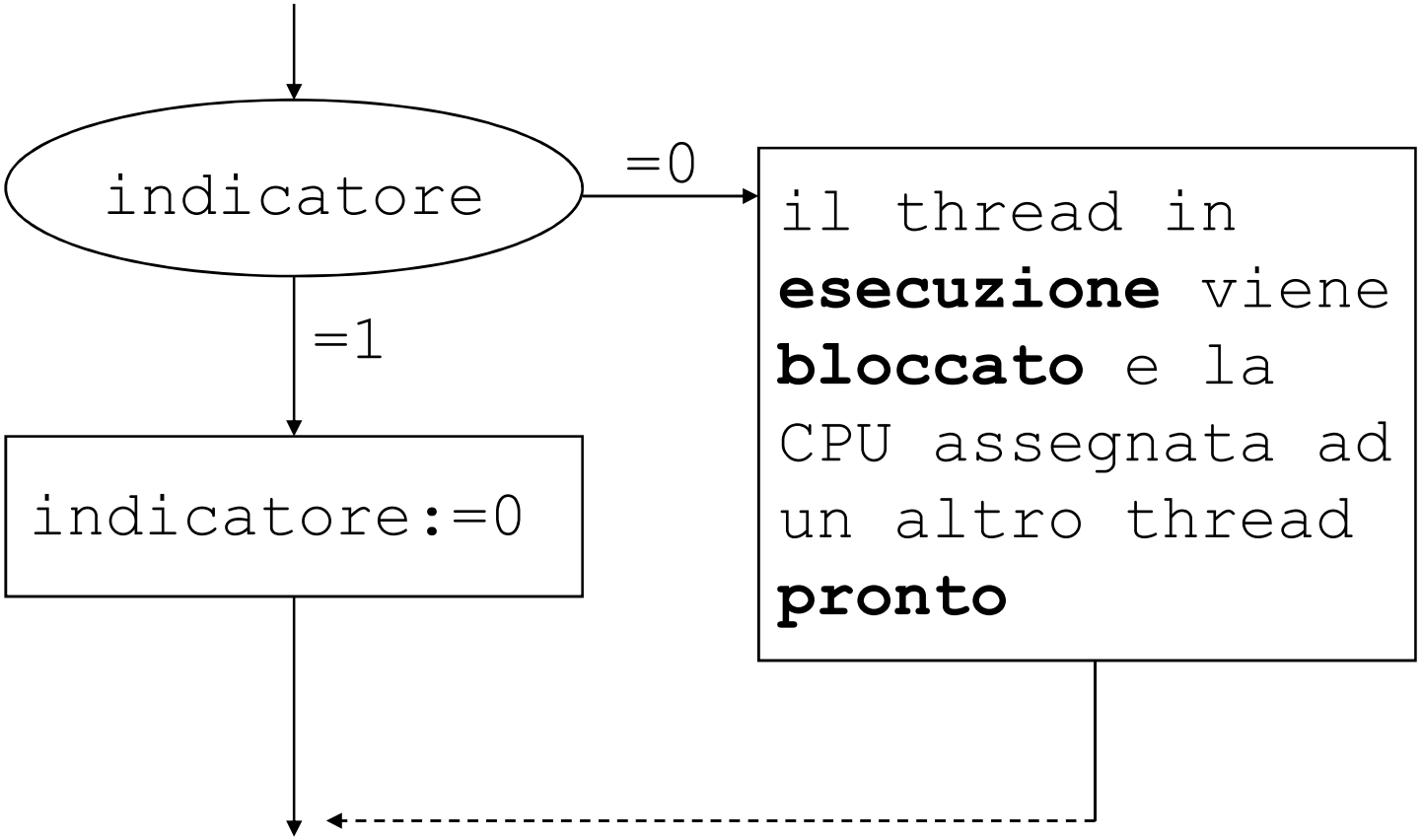
- Idea centrale *per soddisfare anche i requisiti di efficienza*:
  - Utilizzare il concetto di *stato del processo / thread* e il meccanismo di evoluzione degli stati dei thread offerto dal sistema operativo

Per un meccanismo di mutua esclusione generale

- Per rendere *più efficiente* la soluzione e soddisfare *tutti i requisiti* della mutua esclusione nel caso più generale occorre:
  - *bloccare* un thread per tutto il tempo in cui non ha accesso alla sezione critica ($\Rightarrow$ in stato *sospeso* / *waiting*)
  - *riattivarlo* quando, per effetto del progresso di altri thread, il suo accesso alla sezione critica è consentito ($\Rightarrow$ in stato *pronto* o *esecuzione*)
  - riattivare i thread in attesa sulla sezione critica con *una politica che non determini starvation*

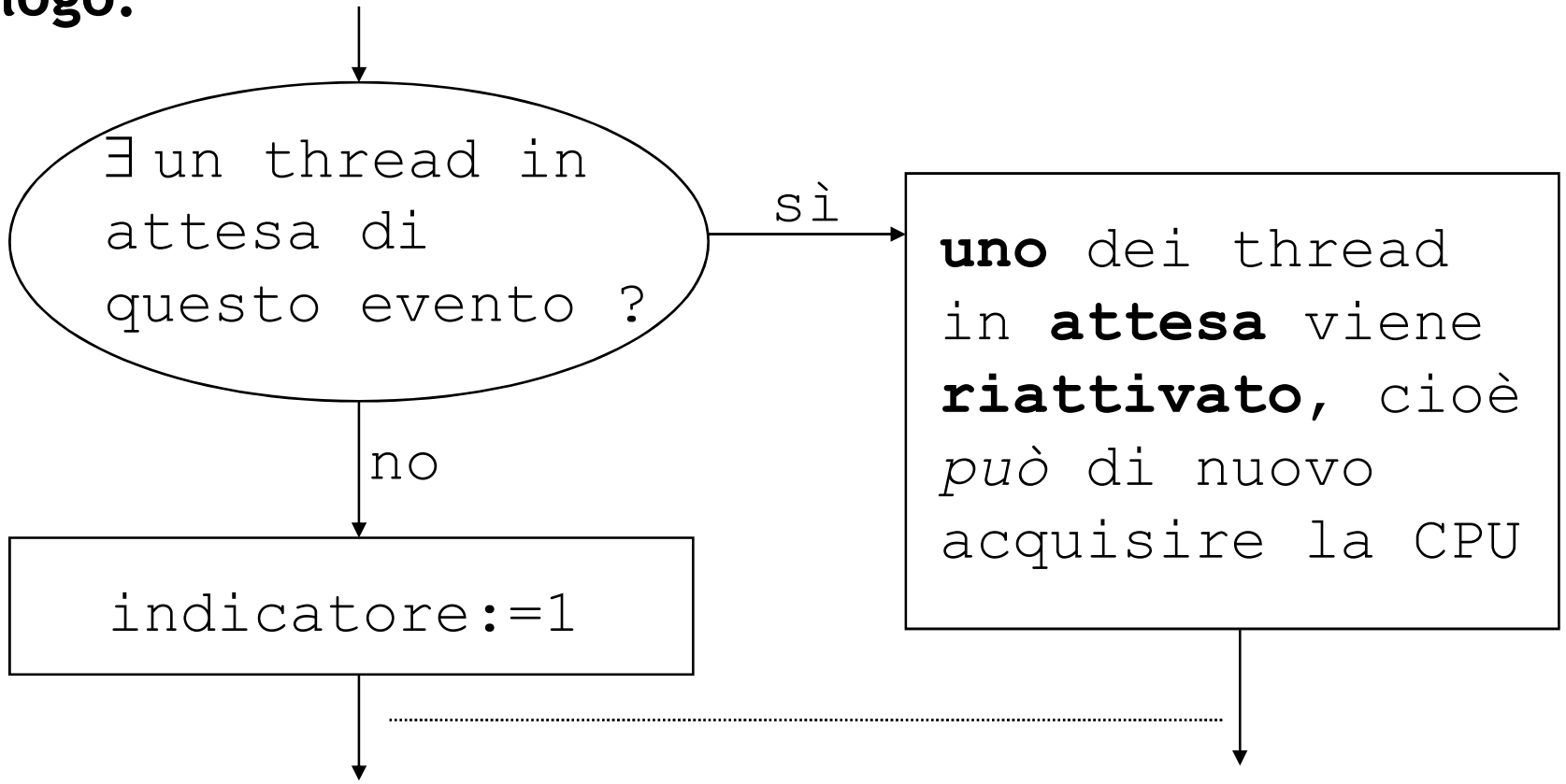
Meccanismo di mutua esclusione generale

- **Prologo:**



Meccanismo di mutua esclusione generale

- **Epilogo:**



Meccanismo di mutua esclusione generale

- Prologo ed Epilogo devono essere operazioni uniche, non divisibili
- Come?

- Sono realizzati come *sezioni critiche brevi* ! ✓

Semafori

- Un semaforo è una *variabile intera non negativa* ($s \geq 0$) con valore iniziale $s_o \geq 0$
- Al semaforo è associata una *coda di attesa* Qs nella quale sono posti i *descrittori* dei processi /thread che attendono l'autorizzazione a proseguire l'esecuzione
- Sul semaforo sono ammesse *solo* due operazioni *indivisibili* (*primitive*): wait(s) e signal(s)

  ovvero: P(s) V(s)

- *wait* e *signal* sono realizzate tramite chiamate al SO (SVC) ed eseguite in modo *monitor*, cioè la variabile semaforo è *protetta*

wait e signal

wait(s):
begin
    if s = 0 *then*
        <il thread viene *sospeso* e
            il suo descrittore inserito in Qs>;
    else s := s - 1;
end;

wait e signal

signal(s):
begin
    if <esiste un thread in coda Qs> *then*
        <il suo descrittore viene rimosso da Qs e
        il suo stato modificato in *pronto*>;
    else s := s + 1;
end;

wait e signal

- La *wait* può essere *passante* (s > 0) o *bloccante* (s = 0), nel qual caso si verifica un *context switch*. La *signal* è sempre *passante*
- L'esecuzione della *signal(s)* non comporta concettualmente alcuna modifica allo stato del thread che l'ha eseguita
  - → Dopo l'esecuzione di una *signal(s)* due thread sono potenzialmente in esecuzione
  - nel caso monoprocessore la definizione di semaforo non specifica chi tra thread segnalante e segnalato prosegue: la realizzazione è considerata comunque logicamente corretta
  - nel caso multicore, entrambi i thread possono essere immediatamente in esecuzione dopo la signal
- La scelta del thread sospeso avviene *tramite politica FIFO*

Mutua esclusione tramite semaforo e primitive wait e signal

- Ad ogni classe di sezioni critiche viene associata una variabile semaforo *s*; prologo ed epilogo vengono realizzati rispettivamente tramite *wait(s)* e *signal(s)*
- A, B sezioni critiche della stessa classe; s semaforo (valore iniziale $s_0 = 1$):

thread T1	thread T2
...	...
wait(s);	*wait(s)*;
<sezione critica A>	<sezione critica B>
signal(s);	*signal(s)*;
...	...

Mutua esclusione tramite semaforo e primitive wait e signal

- La natura primitiva di *wait* e *signal* assicura la proprietà di mutua esclusione
- La soluzione è corretta per *qualunque numero di processi / thread* e per velocità relative arbitrarie
- Sono risolti i problemi di *attesa attiva* e *attesa indefinita* tramite una gestione opportuna della coda dei thread bloccati, (ad esempio FIFO):
  - Un thread non può riappropriarsi della sezione critica che ha appena liberato se ci sono altre richieste pendenti (nella signal è rimasto s = 0)

Indivisibilità di wait e signal

- Occorre garantire che l'azione di analisi e modifica del semaforo non sia separata dalla azione di sospensione
  - Esempio: s = 0

    t0: if s = 0 (T1)

    t1: s := s + 1 (T2)

    t2: sospensione (T1)

    => thread sospeso (T1) su un semaforo che vale 1.

- Si può ottenere indivisibilità *inibendo le interruzioni* durante l'esecuzione di *wait* e *signal*, solo se tutte le *wait* e *signal* relative allo stesso semaforo sono eseguite sullo stesso processore
- Nel caso di sistema multiprocessore occorre considerare *wait* e *signal* come *sezioni critiche brevi* e proteggerle mediante *lock*

Indivisibilità di wait e signal

- Nel caso più generale in cui *wait* e *signal* relative allo stesso semaforo possono essere eseguite su processori diversi la realizzazione può usare sia lock-unlock che disabilitazione delle interruzioni

wait(sem): *begin*
<disabilitazione interruzioni>
lock(x);
<codice della wait>
unlock(x);
<abilitazione interruzioni>
end;

Indivisibilità di wait e signal

signal(sem): *begin*

<disabilitazione interruzioni>

lock(x);

<codice della signal>

unlock(x);

<abilitazione interruzioni>

end;

Livelli di sezioni critiche

- I Livello:

  sezioni critiche: S1, S2

  mutua esclusione tramite wait e signal

- II Livello:

  sezioni critiche: wait(s) e signal(s)

  mutua esclusione tramite lock(x) e unlock(x)

- III Livello:

  sezioni critiche: lock(x), unlock(x)

  mutua esclusione tramite hardware (test-and-set e/o disabilitazione interruzioni)

Relazioni Invarianti

- Poichè *wait* e *signal* sono le uniche operazioni ammesse sulla variabile semaforo, si ha:

$$val(s) = s_0 + ns(s) - nw(s)$$

ove:

val(s)	è il valore del semaforo
s_0	è il valore iniziale
ns(s)	numero di volte che è stata eseguita la signal(s) senza alcun thread in coda
nw(s)	numero di volte che è stata eseguita con successo la wait(s)

Relazioni Invarianti

- Poichè $val(s) \geq 0$ si ha:

$$nw(s) \leq ns(s) + s_0$$

- La relazione è *invariante rispetto alla esecuzione di wait(s) e signal(s)*, cioè è sempre vera qualunque sia il numero di primitive eseguite

Verifiche di correttezza per la soluzione alla mutua esclusione

- *Un solo thread alla volta può trovarsi nella sezione critica*

- Infatti: n = nw(s) - ns(s) (=num. proc. entro la sez. critica)

- L'*invariante* $nw(s) \leq ns(s) + s_0$
  diventa: $nw(s) \leq ns(s) + 1$
  da cui: $n = nw(s) - ns(s) \leq 1$

- Poichè $nw(s) - ns(s) \geq 0$
  si ha: $0 \leq n \leq 1$

Verifiche di correttezza

- *Un thread deve bloccarsi solo se la sezione critica è occupata*
- Infatti, se un thread viene bloccato: $s = 0$
- La relazione $val(s) = ns(s) - nw(s) + s_0$
  diventa: $nw(s) = ns(s) + 1$
- In base a tale relazione il numero delle wait eseguite con successo supera il numero delle signal di 1. Pertanto c'è *un* thread nella sezione critica

Mutua esclusione tramite wait e signal

- La soluzione tramite *wait* e *signal* risolve in maniera *generale* il problema della mutua esclusione
- R1: Le sezioni critiche possono essere eseguite con *interruzioni abilitate*
- R2: Supponiamo per assurdo che *n* thread eseguano contemporaneamente sezioni critiche della stessa classe;

  per come sono fatti prologo ed epilogo si ha:

  $$n = nw(s) - ns(s)$$

  e dalla relazione invariante $nw(s) \leq ns(s) + 1$

  si ha: $0 \leq n \leq 1$

Mutua esclusione tramite wait e signal

- R3: L'*indipendenza nell'accesso alle sezioni critiche* è assicurata dal particolare protocollo usato (*wait*, sezione critica, *signal*) e dal fatto che *wait* e *signal* sono le *uniche* operazioni eseguite su una variabile semaforo
- R4: Se un thread è bloccato sulla *wait(s)* significa che s = 0

  e quindi: val(s) = ns(s) - nw(s) + 1 = 0

  da cui: nw(s) - ns(s) = 1

  cioè c'è necessariamente *un* thread nella sezione critica.
- R5: Il meccanismo non prevede *attese attive*; i thread che operano la *wait* con sezione critica occupata vengono *bloccati*
- R6: Per evitare la starvation è sufficiente gestire in modo opportuno la coda dei thread bloccati (FIFO)

Soluzione generale al problema della mutua esclusione

- Soluzione dal punto di vista del programmatore!
  - (i) alloca un semaforo s con v.i. $s_0 = 1$ per ogni classe di sezioni critiche
  - (ii) realizza prologo ed epilogo tramite *wait(s)* e *signal(s)*

- Es. A, B sezioni critiche della stessa classe:

thread T1	thread T2
...	...
wait(s);	*wait(s)*;
<sezione critica A>	<sezione critica B>
signal(s);	*signal(s)*;
...	...

Quale ruolo per i semafori?

- I meccanismi del SO per mutua esclusione (lock/unlock e/o semafori) sono variamente incapsulati da linguaggi e librerie, … ovvero …
- Semafori e primitive di sincronizzazione sono realizzati dal linguaggio di programmazione o dalle librerie con funzionalità di base offerte dal SO

- Primitive wait() e signal():
  - Potenzialità
  - Limiti
  - Pregi e difetti